Scrabble®

Versión 1.1

Alexandre Vinent Padrol
Soukaïna Mahboub Mehboub

Índice

1. Introducción

Scrabble® Versión 1.1 es una aplicación desarrollada en el marco de la asignatura *Proyectos de Programación*. El objetivo es trasladar al entorno digital la experiencia del juego de mesa Scrabble, siguiendo fielmente sus reglas oficiales. El proyecto está implementado íntegramente en Java, utilizando una arquitectura basada en capas bien diferenciadas: presentación, dominio y persistencia.

La capa de dominio contiene tanto las clases que modelan los elementos fundamentales del juego, como los controladores que encapsulan la lógica de negocio: ***CtrlDominio***, ***CtrlJugador*** y ***CtrlPartida***. El ***CtrlDominio*** actúa como punto central de coordinación, y se comunica exclusivamente con los controladores ***CtrlJugador*** y ***CtrlPartida*** para gestionar la lógica de las funcionalidades principales del sistema.

Siguiendo los principios de diseño de software en capas, la capa de presentación se ha desacoplado completamente de la lógica interna. En esta entrega, dicha capa está representada por ***drivers*** que interactúan únicamente con el ***CtrlDominio***, respetando así el principio de inversión de dependencias. Estos drivers incluyen menús de prueba que permiten al usuario realizar acciones como registrar usuarios, iniciar sesión, comenzar partidas o consultar el ranking global, facilitando el testeo y validación del sistema.

La capa de persistencia ha sido diseñada con un enfoque progresivo. Por el momento, se ha implementado una persistencia mixta: los datos de usuarios y el ranking (con sus respectivas puntuaciones) se guardan físicamente en "disco" mediante ficheros .txt, mientras que las partidas activas se gestionan mediante estructuras de datos dentro del controlador de persistencia. Esta aproximación es para avanzar en el desarrollo manteniendo flexibilidad para futuras integraciones de almacenamiento.

El controlador de dominio es también responsable de coordinar el acceso a los datos persistentes, gestionando tanto la recuperación como la actualización de la información a través del controlador de persistencia.

La estructura modular del sistema es para garantizar su mantenibilidad y escalabilidad. Las relaciones entre las entidades del modelo de dominio, se reflejan visualmente en el diagrama de clases UML incluido en este documento, que facilitará la comprensión global del diseño.

Hay cuatro drivers independientes **DriverGestionUsuarios**, **DriverGestionDiccionariosBolsas**, **DriverGestionPartidas** y **DriverGestionJuego** incorporan cada uno, dentro de su propio menú de consola, un conjunto de juegos de prueba. Por su parte, **DriverScrabble** actúa como un driver único e integral que ofrece todas esas operaciones de forma conjunta, permitiendo recorrer de principio a fin todos los casos de uso sin necesidad de cambiar de programa.

2. Casos de uso del Scrabble

2.1. Diagrama de casos de uso

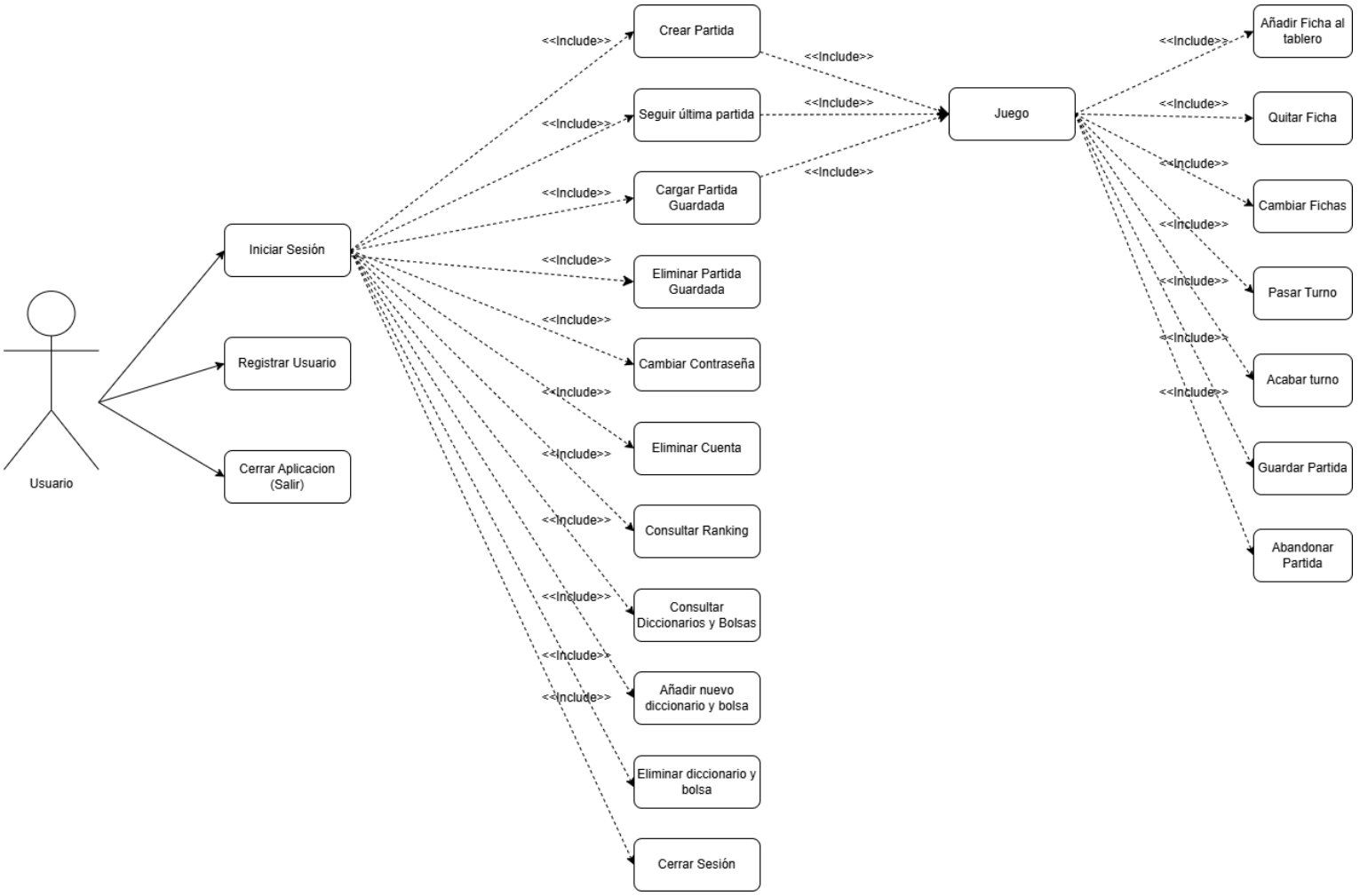

fig1. Diagrama de casos de uso del Scrabble implementado.

El diagrama de casos de uso en mejor calidad se puede encontrar en formato PDF en la carpeta DOCS del proyecto entregado.

2.2. Descripción detallada de los casos de uso

2.2.1. Gestión de usuario

2.2.1.1. Registrar Jugador

Actor principal: Usuario
Precondición: El usuario no debe estar registrado aún con ese nombre.
Activador: El usuario selecciona "Registrarse" desde el menú principal.
Escenario de éxito principal:

- El usuario introduce un nombre de usuario y una contraseña.
- El sistema registra al nuevo jugador con 0 puntos.
- Se inicia sesión automáticamente después del registro.

Errores y excepciones:

- Si el nombre de usuario ya está registrado:
  Se lanza **UsuarioYaRegistradoException**.

2.2.1.2. Iniciar sesión

Actor principal: Usuario
Precondición: El usuario debe estar registrado previamente.
Activador: El usuario selecciona "Iniciar Sesión" en el menú principal.
Escenario de éxito principal:

- El usuario introduce nombre de usuario y contraseña.
- El sistema verifica que el usuario existe y que la contraseña es correcta.
- Se inicia sesión y se muestran sus puntos y posición en el ranking.

Errores y excepciones:

- Si el usuario no existe:

  Se lanza **UsuarioNoEncontradoException**.
- Si la contraseña es incorrecta:

  Se lanza **PasswordInvalidaException**.

2.2.1.3. Cerrar aplicación (Salir)

Actor principal: Usuario
Precondición: La aplicación está iniciada.
Activador: El usuario elige "Salir" en cualquiera de los menús.

Escenario de éxito principal

- En el **Menú Principal**, el usuario selecciona la opción **"3. Salir"**, o bien,
- En el **Menú Usuario**, el usuario selecciona la opción **"4. Salir"**.
- El sistema ejecuta internamente System.exit(0) y finaliza la aplicación de inmediato.

Errores y extensiones

- No aplica: la aplicación se detiene sin más, acabando con la ejecución del proceso.

2.2.1.4. Cambiar contraseña

Actor principal: Usuario
Precondición: El usuario está autenticado.
Activador: El usuario selecciona "Cambiar Contraseña".
Escenario de éxito principal:

- El sistema solicita la contraseña actual y la nueva contraseña (dos veces).
- Verifica que ambas nuevas coincidan y que la actual sea válida.
- Si es correcto, actualiza la contraseña.

Se muestra:
"Contraseña cambiada correctamente."

Errores y excepciones:

- Si las nuevas contraseñas no coinciden: Se muestra "Error: las contraseñas no coinciden."
- Si la contraseña actual es incorrecta:
  Se lanza **PasswordInvalidaException**.

2.2.1.5. Eliminar cuenta

Actor principal: Usuario
Precondición: El usuario está autenticado.
Activador: El usuario selecciona "Eliminar Perfil".
Escenario de éxito principal:

- El sistema pide confirmación y solicita la contraseña.
- Si es válida, elimina el perfil del usuario.

Muestra:
“Perfil eliminado. Adiós.”

Errores y excepciones:

- Si la contraseña es incorrecta: Se lanza **PasswordInvalidaException**.
- Si el usuario cancela la confirmación: Se muestra “Eliminación cancelada.”

2.2.1.6. Cerrar sesión

Actor principal: Usuario
Precondición: El usuario está autenticado.
Activador: El usuario selecciona "Cerrar sesión".
Escenario de éxito principal:

- El sistema finaliza la sesión actual.

Se muestra:
“Sesión cerrada.”

2.2.1.7. Consultar ranking

Actor principal: Usuario
Precondición: El usuario está autenticado.
Activador: El usuario selecciona "Ver Ranking".
Escenario de éxito principal:

- El sistema obtiene y muestra el ranking global de jugadores.

Errores y excepciones:

- Si no hay jugadores en el ranking:
  Se lanza **RankingVacioException.**

2.2.2. Gestión de partidas

2.2.2.1. Crear partida

Actor principal: Usuario
Precondición: El usuario está en el menú principal.
Activador: El usuario selecciona "Nueva Partida".
Escenario de éxito principal:
El sistema solicita al usuario los siguientes pasos:

- Selección del modo de juego (1 o 2 jugadores).
- Selección de un diccionario.
- Introducción del nombre del jugador 1 (y del jugador 2 si aplica).
- El sistema valida los datos introducidos.
- El sistema inicia una nueva partida con una semilla aleatoria.
- Se muestra el tablero y se accede al menú de partida.

Errores y excepciones:

- Si faltan datos: Se notifica con "¡Faltan datos! Completa modo, diccionario y nombres."
- Si el diccionario no se encuentra:
  Lanza **DiccionarioNoEncontradoException**.
- Si la bolsa no se encuentra:
  Lanza **BolsaNoEncontradaException**.

2.2.2.2. Seguir última partida

Actor principal: Usuario
Precondición: Existe una partida guardada previamente.
Activador: El usuario selecciona "Seguir última partida".
Escenario de éxito principal:

- El sistema carga la última partida guardada.
- El sistema muestra el tablero y accede al menú de partida.

Errores y excepciones:

- Si no existe una partida previa:
  Lanza **NoHayPartidaGuardadaException**.

2.2.2.3. Cargar partida guardada

Actor principal: Usuario
Precondición: El usuario está en el menú principal.
Activador: El usuario selecciona "Cargar Partida".
Escenario de éxito principal:

- El sistema muestra una lista de partidas guardadas.
- El usuario elige una partida por número o por ID.
- El sistema carga la partida seleccionada y accede al menú de partida.

Errores y excepciones:

- Si no hay partidas guardadas se lanza **NoHayPartidaGuardadaException.**
- Si se introduce una selección inválida: Se lanza con **PartidaNoEncontradaException**.

2.2.2.4. Guardar partida

Actor principal: Usuario
Precondición: El usuario está dentro de una partida.
Activador: El usuario elige "Guardar partida".
Escenario de éxito principal:

- El sistema solicita un nombre.
- El usuario introduce el nombre.
- El sistema guarda la partida bajo ese nombre.

Errores y excepciones:

- Si no se introduce un nombre: Se muestra “Nombre inválido.”
- Si se introduce un nombre ya existente, se lanza **PartidaYaExisistnteException**.

2.2.2.5. Abandonar partida

Actor principal: Usuario
Precondición: El usuario está dentro de una partida.
Activador: El usuario selecciona "Abandonar".
Escenario de éxito principal:
El sistema sale de la partida actual y vuelve al menú principal.

2.2.2.6. Eliminar partida guardada

Actor principal: Usuario
Precondición: El usuario accede a "Gestión partidas guardadas" y selecciona una partida válida.
Activador: El usuario elige la opción "Eliminar" tras seleccionar una partida.
Escenario de éxito principal:

- El usuario introduce el ID a eliminar
- El sistema elimina la partida seleccionada del almacenamiento.
- Se muestra un mensaje confirmando la eliminación.

Errores y excepciones:

- Si introduce un ID incorrecto se notifica con: “Selección fuera de rango.”
- Se notifica con un mensaje de error apropiado (por ejemplo, “No se pudo eliminar la partida”).
- Si la partida no existe, se lanzs **PartidaNoEncontradaException**

2.2.3. Gestión de juego

2.2.3.1. Añadir ficha en el tablero

Actor principal: Usuario
Precondición: El usuario se encuentra en el menú de partida.
Activador: El usuario selecciona la opción "Añadir ficha".
Escenario de éxito principal:

- El sistema muestra el tablero y las fichas disponibles.
- El usuario introduce la letra y posición en el formato adecuado.
- El sistema ejecuta cd.jugarScrabble(1, input) para colocar la ficha.
- Se actualiza el tablero.

Errores y excepciones:

- Si el input está mal formado: se lanza **ComandoInvalidoException**.
- Si la palabra resultante no es válida: se lanza **PalabraInvalidaException**.

2.2.3.2. Quitar ficha del tablero

Actor principal: Usuario
Precondición: El usuario se encuentra en el menú de partida.
Activador: El usuario selecciona la opción "Quitar ficha".
Escenario de éxito principal:

- El sistema muestra el estado del tablero.
- El usuario introduce la posición a quitar.
- El sistema ejecuta cd.jugarScrabble(2, input).
- Se actualiza el tablero.

Errores y excepciones:

- Posición inválida o malformada: **ComandoInvalidoException**.

2.2.3.3. Pasar turno

Actor principal: Usuario
Precondición: El usuario se encuentra en el menú de partida.
Activador: El usuario selecciona la opción "Pasar turno".
Escenario de éxito principal:

- El sistema ejecuta cd.jugarScrabble(3, "").
- Se transfiere el turno al siguiente jugador (IA o humano).
- Se muestra el nuevo estado del tablero y fichas.

Errores y excepciones:

- No aplica

2.2.3.4. Finalizar turno

Actor principal: Usuario
Precondición: El usuario se encuentra en el menú de partida y ha colocado fichas.
Activador: El usuario selecciona la opción "Finalizar turno".
Escenario de éxito principal:

- El sistema ejecuta cd.jugarScrabble(4, "").
- Se validan y aplican las jugadas en el tablero.
- Se actualizan los puntos del jugador.
- Se transfiere el turno al siguiente jugador.

Errores y excepciones:

- Jugada inválida al poner una palabra incorrecta: **PalabraInvalidaException**.

2.2.3.5. Cambiar fichas

Actor principal: Usuario
Precondición: El usuario está en el menú de partida.
Activador: El usuario selecciona la opción "Cambiar fichas".
Escenario de éxito principal:

- El sistema solicita las fichas a cambiar.
- El usuario introduce las fichas.
- El sistema ejecuta cd.jugarScrabble(5, input).
- Las fichas se intercambian y se actualiza las fichas disponibles del jugador.

Errores y excepciones:

- Entrada incorrecta o formato inválido al introducir las fichas: **ComandoInvalidoException**.

2.2.4. Gestión Diccionarios y Bolsas

2.2.4.1. Consultar Diccionarios y Bolsas

Actor principal: Usuario
Precondición: El sistema ha iniciado correctamente.
Activador: El usuario selecciona la opción “Consultar diccionarios y bolsas” en el menú principal.

Escenario de éxito principal:

- El sistema recoge los identificadores de diccionarios y bolsas disponibles.
- Se muestran en pantalla en una tabla, indicando cuáles existen por cada

Errores y excepciones:

- Si no hay diccionarios ni bolsas, se muestra:
  “ No hay entradas registradas.”
- El sistema regresa al menú principal.

2.2.4.2. Añadir nuevo diccionario y bolsa

Actor principal: Usuario
Precondición: El usuario accede a la opción de gestión de diccionarios y bolsas..
Activador: El usuario elige añadir un recurso (diccionario + bolsa) desde un directorio.

Escenario 1: Desde directorio existente

- El usuario selecciona "Desde directorio existente".
- El sistema solicita una ruta de directorio.
- El sistema verifica la existencia de los archivos:
    - **ID_diccionario.txt**
    - **ID_bolsa.txt**
- Si están presentes y válidos:
    - El sistema carga las palabras del diccionario y la configuración de la bolsa.
    - Se registra el recurso.
    - Se muestra: “Recurso 'ID' añadido desde directorio.”

Errores y excepciones:

- Directorio no válido:
  “No es un directorio válido.”
- Archivos requeridos ausentes:
  “Faltan archivos en el directorio.”
- Excepción de entrada/salida (IOException):
  “Error I/O: mensaje del sistema”

Escenario 2: Desde entrada por teclado

- El usuario selecciona "Desde teclado".
- El sistema solicita:
    - ID del recurso
    - Palabras (una por línea, en mayúsculas)
    - Líneas de bolsa con formato: "Letra Repeticiones Puntos"
- El sistema construye el diccionario y la bolsa a partir de los datos.
- Se crea y registra el recurso.
- Se muestra: “Recurso 'ID' creado correctamente.”

Errores y excepciones:

- Excepción de entrada/salida (IOException):
  “Error al crear recurso: mensaje del sistema”
- DiccionarioYaExistenteException:
  “Ya existe un diccionario con ese ID.”
- BolsaYaExistenteException:
  “Ya existe una bolsa con ese ID.”

2.2.4.3. Eliminar diccionario y bolsa

Actor principal: Usuario
Precondición: Existen diccionarios o bolsas creados.
Activador: El usuario selecciona la opción “Eliminar diccionario+bolsa”.
Escenario de éxito principal:

- El usuario introduce un ID.
- El sistema elimina el diccionario y la bolsa correspondientes.
- Muestra:
  “Recurso 'ID' eliminado.”

Errores y excepciones:

- **IOException** al eliminar. Si ocurre un error al eliminar el recurso desde el sistema de archivos: “Error al eliminar recurso: *mensaje*”
- **DiccionarioNoEncontradoException / BolsaNoEncontradaException:** Estas excepciones pueden ser lanzadas si alguno de los recursos no existe. En el código no se manejan con un mensaje específico, pero podrían causar un fallo del programa si no se controlan desde el main().
- Si el usuario introduce “0”, se cancela y vuelve al menú.

3. Modelo conceptual

3.1. Diagrama del modelo conceptual

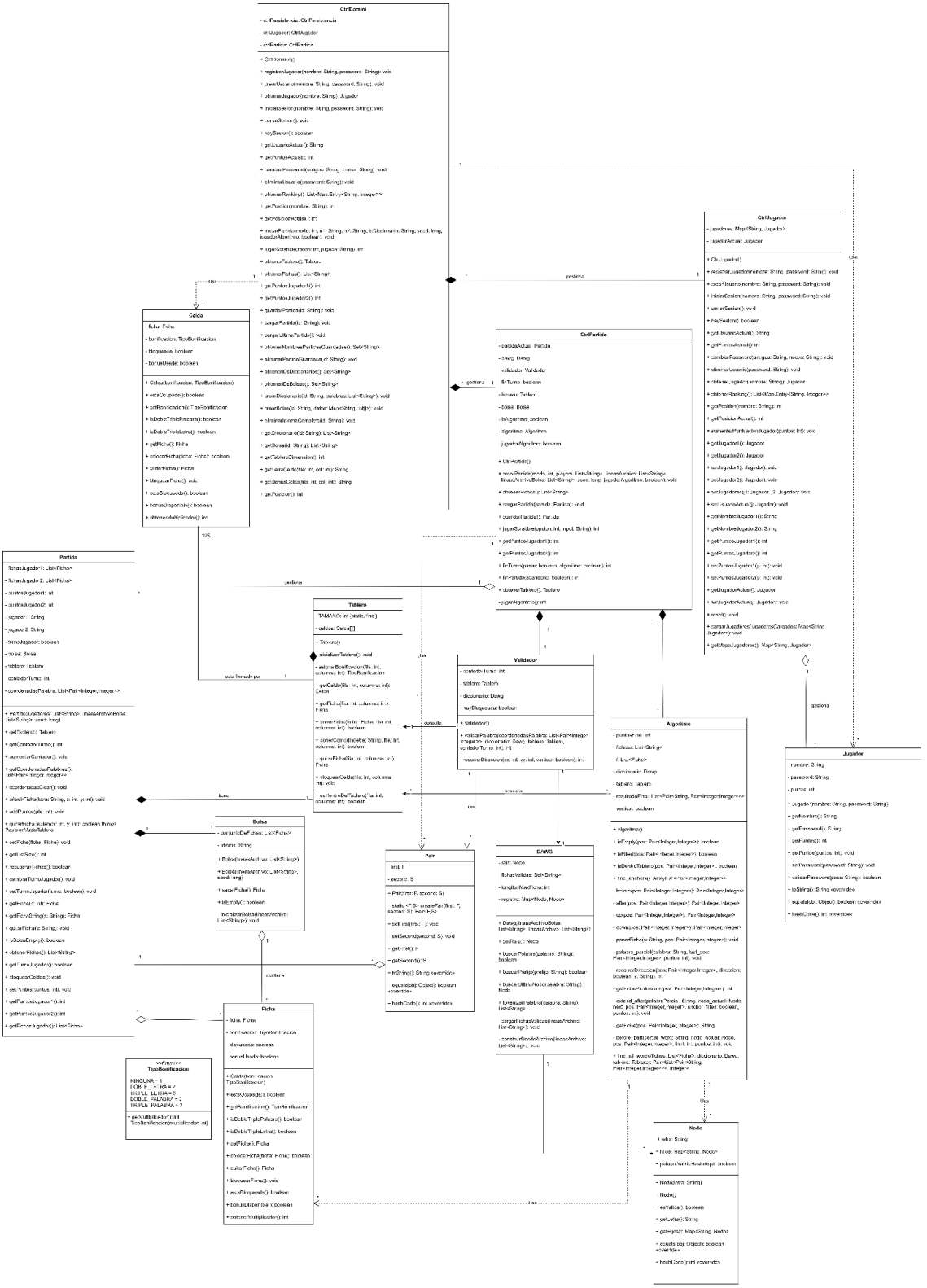

fig2. Diagrama del modelo conceptual (Capa Dominio) del Scrabble implementado (Se encuentra en la carpeta DOCS).

3.2. Descripción detallada de las clases

3.2.1. Modelos

3.2.1.1. Ficha

Nombre de la clase Ficha

Breve descripción de la clase
Representa una ficha de Scrabble con su letra (o símbolo) y puntuación asociada.

Descripción de los atributos

- **id** (int): identificador único de la ficha (opcional).
- **letra** (String): letra o símbolo que representa la ficha.
- **puntuacion** (int): puntos que vale la ficha al colocarse.

Descripción de los métodos

- **Ficha(String letra, int puntuacion)**: constructor que asigna letra y puntuación.
- **String getLetra()**: devuelve la letra o símbolo de la ficha.
- **int getPuntuacion()**: devuelve los puntos de la ficha.
- **String toString()**: representación en cadena "letra(puntuacion)", por ejemplo "A(1)".
- **boolean equals(Object obj)**: true si otro objeto es Ficha con misma letra y puntuación.
- **int hashCode()**: código hash basado en letra y puntuación.

3.2.1.2. Bolsa

Nombre de la clase Bolsa

Breve descripción de la clase
Bolsa de fichas para Scrabble que se inicializa desde líneas de configuración, se baraja (con o sin semilla) y permite extraer fichas aleatoria o secuencialmente.

Descripción de los atributos

- **conjuntoDeFichas** (List<Ficha>): lista de fichas disponibles en la bolsa.
- **idioma** (String): código de idioma asociado.

Descripción de los métodos

- **Bolsa(List<String> lineasArchivo)**: carga fichas desde las líneas dadas y baraja aleatoriamente.
- **Bolsa(List<String> lineasArchivo, long seed)**: como el anterior, pero usa la semilla para barajar de forma reproducible.
- **void inicializarBolsa(List<String> lineasArchivo)**: parsea cada línea "Letra Cantidad Puntuacion" y añade las fichas correspondientes a la lista.
- **String getIdioma()**: devuelve el código de idioma de la bolsa.
- **Ficha sacarFichaAleatoria()**: extrae y elimina una ficha al azar; o null si está vacía.
- **Ficha sacarFicha()**: extrae y elimina la primera ficha; o null si está vacía.
- **boolean isEmpty()**: true si no quedan fichas; false en caso contrario.
- **int size()**: número de fichas restantes en la bolsa.

3.2.1.3. TipoBonificacion

Nombre de la clase TipoBonificacion

Breve descripción de la clase
Enumeración de las bonificaciones de Scrabble, cada una con su multiplicador para letra o palabra.

Descripción de los atributos

- **multiplicador** (int): valor que se usa para multiplicar la puntuación según el tipo de bonificación.

Descripción de los métodos

- **TipoBonificacion(int multiplicador)**: constructor interno que asigna el multiplicador a la constante.

- **int getMultiplicador()**: devuelve el multiplicador asociado a la bonificación.

3.2.1.4. Celda

Nombre de la clase Celda

Breve descripción de la clase
Casilla individual del tablero de Scrabble que puede contener una ficha, tener una bonificación de letra o palabra, y llevar el estado de bloqueo de la ficha y uso de la bonificación.

Descripción de los atributos

- **ficha (Ficha)**: ficha presente en la celda, o null si está vacía.
- **bonificacion (TipoBonificacion):** tipo de bonificación asignada (doble/triple letra/palabra o ninguna).
- **bloqueada (boolean):** indica si la ficha está bloqueada (no extraíble).
- **bonusUsada (boolean):** señala si ya se usó la bonificación de la celda.

Descripción de los métodos

- **Celda(TipoBonificacion bonificacion)**: inicializa celda vacía, no bloqueada y con bonificación disponible.
- **boolean estaOcupada()**: devuelve true si hay ficha; false si está vacía.
- **TipoBonificacion getBonificacion()**: retorna el tipo de bonificación.
- **boolean isDobleTriplePalabra()**: true si bonificación es DOBLE_PALABRA o TRIPLE_PALABRA.
- **boolean isDobleTripleLetra()**: true si bonificación es DOBLE_LETRA o TRIPLE_LETRA.
- **Ficha getFicha()**: devuelve la ficha actual, o null si no hay.
- **boolean colocarFicha(Ficha ficha)**: coloca ficha si celda libre y no bloqueada; retorna éxito.
- **Ficha quitarFicha()**: quita ficha si no está bloqueada; retorna ficha o null.
- **void bloquearFicha()**: bloquea la ficha y marca la bonificación como usada.
- **boolean estaBloqueada()**: true si la ficha está bloqueada.
- **boolean bonusDisponible()**: true si la bonificación aún no fue usada.
- **int obtenerMultiplicador()**: retorna bonificacion.getMultiplicador() si disponible; 1 en caso contrario.

3.2.1.5. Tablero

Nombre de la clase Tablero

Breve descripción de la clase
Tablero 15×15 de Scrabble con celdas que pueden contener fichas y bonificaciones.

Descripción de los atributos

- **TAMANO** (static final int): tamaño fijo (15).
- **celdas (Celda[][])**: matriz de 15×15 celdas.

Descripción de los métodos

- **Tablero()**: crea la matriz y llama a inicializarTablero().
- **inicializarTablero()**: recorre i,j y asigna a celdas[i][j] una Celda(asignarBonificacion(i,j)).
- **asignarBonificacion(int fila,int col)**: devuelve el TipoBonificacion según coordenadas; por defecto NINGUNA.
- **getCelda(int fila,int col)**: devuelve la Celda si índices válidos; si no, null.
- **getFicha(int fila,int col)**: devuelve la ficha de la celda o null.
- **ponerFicha(Ficha ficha,int fila,int col)**: coloca ficha si la celda existe y está libre; devuelve boolean.
- **ponerComodin(String letra,int fila,int col)**: crea ficha comodín y la coloca; devuelve boolean.
- **quitarFicha(int fila,int col)**: quita y retorna ficha si no bloqueada; si no, null.
- **bloquearCelda(int fila,int col)**: bloquea ficha y marca la bonificación usada.
- **esCentroDelTablero(int fila,int col)**: true si la posición es (7,7); false en otro caso.

3.2.1.6. Jugador

Nombre de la clase Jugador

Breve descripción de la clase
Modelo de jugador con nombre único, contraseña y puntuación máxima alcanzada.

Descripción de los atributos

- **nombre** (String): nombre único del jugador.
- **password** (String): contraseña para autenticar al jugador.
- **puntos** (int): puntuación máxima alcanzada.

Descripción de los métodos

- **Jugador(String nombre, String password)**: constructor que asigna nombre, contraseña y deja puntos = 0.
- **String getNombre()**: devuelve el nombre del jugador.
- **String getPassword()**: devuelve la contraseña actual.
- **int getPuntos()**: devuelve la puntuación máxima.
- **void setPuntos(int puntos)**: actualiza la puntuación máxima.
- **void setPassword(String password)**: cambia la contraseña.
- **boolean validarPassword(String pass)**: true si pass coincide con la contraseña; false en otro caso.
- **String toString()**: representación "Jugador{name='…', puntos=…}".
- **boolean equals(Object obj)**: true si otro Jugador tiene el mismo nombre.
- **int hashCode()**: código hash basado en nombre.

3.2.1.7. Partida

Nombre de la clase Partida

Breve descripción de la clase
Gestiona una partida de Scrabble entre dos jugadores: sus manos, puntuaciones, turno, bolsa, tablero y coordenadas de la palabra en juego.

Descripción de los atributos

- **fichasJugador1**, **fichasJugador2** (List<Ficha>): fichas en mano de cada jugador.
- **puntosJugador1**, **puntosJugador2** (int): puntuaciones acumuladas.
- **jugador1**, **jugador2** (String): nombres de los jugadores.
- **turnoJugador** (boolean): true = turno de jugador1, false = de jugador2.
- **bolsa** (Bolsa): bolsa común de fichas.
- **tablero** (Tablero): tablero de la partida.
- **contadorTurno** (int): número de turnos jugados.
- **coordenadasPalabra** (List<Pair<Integer,Integer>>): posiciones de la palabra en curso.

Descripción de los métodos

- **Partida(List<String> jugadores, List<String> lineasArchivoBolsa, long seed)**: inicializa bolsa (con/sin semilla), reparte 7 fichas y crea el tablero.
- **Tablero getTablero()**: devuelve el tablero.
- **int getContadorTurno()**: devuelve el contador de turnos.
- **void aumentarContador()**: incrementa contadorTurno.
- **List<Pair<Integer,Integer>> getCoordenadasPalabras()**: retorna coordenadas de la palabra actual.
- **void coordenadasClear()**: limpia coordenadasPalabra.

- **void añadirFicha(String letra, int x, int y)**: coloca ficha o comodín en (x,y) y registra la posición.
- **void addPuntos(int pts)**: suma puntos al jugador actual.
- **boolean quitarFichaTablero(int x, int y)**: retira ficha del tablero y la devuelve a la mano; lanza excepción si está vacía.
- **void setFicha(Ficha ficha)**: añade ficha a la mano del jugador actual.
- **int getListSize()**: devuelve número de fichas en mano del jugador actual.
- **boolean recuperarFichas()**: rellena manos hasta 7 ficha; devuelve false si la bolsa se vacía.
- **void cambiarTurnoJugador() / void setTurnoJugador(boolean turno)**: alterna o fija el turno.
- **Ficha getFicha(int n)**: obtiene la ficha en posición n de la mano actual.
- **Ficha getFichaString(String s)**: busca ficha por letra en la mano actual.
- **void quitarFicha(String s)**: elimina la primera ficha con letra s de la mano actual.
- **boolean isBolsaEmpty()**: true si la bolsa está vacía.
- **List<String> obtenerFichas()**: lista de letras en mano actual.
- **boolean getTurnoJugador()**: indica quién tiene el turno.
- **void bloquearCeldas()**: bloquea en el tablero las celdas usadas este turno.
- **void setPuntos(int puntos)**: fija la puntuación del jugador actual.
- **int getPuntosJugador1() / int getPuntosJugador2()**: obtienen la puntuación de cada jugador.
- **List<Ficha> getFichasJugador()**: devuelve la lista de fichas del jugador actual.

3.2.1.8. Validador

Nombre de la clase Validador

Breve descripción de la clase
Valida jugadas en el tablero según reglas de Scrabble (alineación, conexión, centro en primer turno, diccionario DAWG) y calcula la puntuación de palabra principal y perpendiculares.

Descripción de los atributos

- **contadorTurno** (int): índice del turno actual.
- **tablero** (Tablero): estado del tablero de la partida.
- **diccionario** (Dawg): estructura DAWG con palabras válidas.
- **hayBloqueada** (boolean): indica si alguna ficha nueva quedó bloqueada en la validación.

Descripción de los métodos

- **Validador()**: constructor por defecto.
- **int validarPalabra(List<Pair<Integer,Integer>> coordenadasPalabra, Dawg diccionario, Tablero tablero, int contadorTurno)**: valida alineación, uso del centro, conexiones, consulta DAWG y suma puntos; retorna puntuación total o 0/–1 según regla.
- **private int recorrerDireccion(int xx, int yy, boolean vertical)**: recorre horizontal o vertical desde (xx,yy), arma palabra, aplica bonificaciones y devuelve su puntuación (o 1 si inválida).

3.2.1.9. Algoritmo

Nombre de la clase Algoritmo

Breve descripción de la clase
Clase que busca y construye jugadas posibles de Scrabble usando las fichas del jugador, el tablero y un diccionario DAWG, calcula la puntuación de cada opción y selecciona la mejor jugada.

Descripción de los atributos

- **puntosFinal** (int): mejor puntuación encontrada hasta el momento.
- **fichass** (List<String>): lista de letras disponibles (incluye comodines '#').
- **f** (List<Ficha>): objetos Ficha correspondientes a las letras del jugador.
- **diccionario** (Dawg): estructura DAWG con las palabras válidas.
- **tablero** (Tablero): estado actual del tablero.
- **resultadoFinal** (List<Pair<String,Pair<Integer,Integer>>>): jugada óptima como lista de pares (letra,posición).
- **vertical** (boolean): orientación de búsqueda (false = horizontal, true = vertical).

Descripción de los métodos

- **Algoritmo()**: constructor por defecto.
- **boolean isEmpty(Pair<Integer,Integer> pos)**: devuelve true si la celda en pos existe y está vacía.
- **boolean isFilled(Pair<Integer,Integer> pos)**: devuelve true si la celda en pos existe y está ocupada.
- **boolean isDentroTablero(Pair<Integer,Integer> pos)**: devuelve true si pos está dentro de los límites del tablero.
- **ArrayList<Pair<Integer,Integer>> find_anchors()**: devuelve lista de posiciones ancla (celdas vacías adyacentes a una ocupada).
- **Pair<Integer,Integer> before(Pair<Integer,Integer> pos)**: posición anterior a pos según la orientación actual.

- **Pair<Integer,Integer> after(Pair<Integer,Integer> pos)**: posición siguiente a pos según la orientación actual.

- **Pair<Integer,Integer> up(Pair<Integer,Integer> pos)**: posición superior a pos.

- **Pair<Integer,Integer> down(Pair<Integer,Integer> pos)**: posición inferior a pos.

- **void ponerFicha(String s, Pair<Integer,Integer> pos)**: añade la letra s en pos a resultadoFinal.

- **void palabra_parcial(String palabra, Pair<Integer,Integer> last_pos, int puntos)**: evalúa una palabra parcial terminada en last_pos, calcula su puntuación (bonificaciones y bingo) y actualiza resultadoFinal si supera puntosFinal.

- **int recorrerDireccion(Pair<Integer,Integer> pos, boolean direccion, String s)**: construye y puntúa la palabra resultante al colocar s en pos (recorrido bidireccional), devuelve puntuación o –1 si no válida.

- **int getFichaPuntuacion(Pair<Integer,Integer> pos)**: devuelve los puntos de la ficha en pos o 0 si está vacía.

- **void extend_after(String palabraParcial, Nodo nodo_actual, Pair<Integer,Integer> next_pos, boolean anchor_filled, int puntos)**: extiende palabraParcial hacia adelante desde next_pos según el DAWG, probando letras disponibles y generando llamadas recursivas.

- **String getFicha(Pair<Integer,Integer> pos)**: devuelve la letra de la ficha en pos.

- **void before_part(String partial_word, Nodo nodo_actual, Pair<Integer,Integer> pos, int limit, int puntos)**: explora prefijos hacia atrás hasta limit, llamando a extend_after para cada rama.

- **Pair<List<Pair<String,Pair<Integer,Integer>>>, Integer> find_all_words(List<Ficha> fichas, Dawg diccionario, Tablero tablero)**: punto de entrada que inicializa estado, obtiene anclas y, para cada orientación, genera y evalúa todas las jugadas posibles; devuelve la jugada óptima y su puntuación.

3.2.1.10. Dawg

Nombre de la clase Dawg

Breve descripción de la clase
Estructura DAWG (Directed Acyclic Word Graph) para cargar un listado de palabras y fichas válidas, ofrecer tokenización eficiente y operaciones de búsqueda de palabra, prefijo y obtención de nodos.

Descripción de los atributos

- **raiz** (Nodo): nodo raíz del DAWG.
- **fichasValidas** (Set<String>): conjunto de fichas (letras o grupos) permitidas para tokenizar.
- **longitudMaxFicha** (int): longitud máxima de ficha en la tokenización.
- **registro** (Map<Nodo,Nodo>): mapa auxiliar para minimización de nodos equivalentes.

Descripción de los métodos

- **Dawg(List<String> lineasArchivoBolsa, List<String> lineasArchivo)**: constructor que inicializa la raíz, carga fichas válidas y construye el DAWG a partir de palabras ordenadas.
- **Nodo getRaiz()**: devuelve el nodo raíz del DAWG.
- **private void cargarFichasValidas(List<String> lineasArchivo)**: extrae fichas válidas de las líneas dadas y ajusta longitudMaxFicha.
- **public List<String> tokenizarPalabra(String palabra)**: tokeniza de forma "greedy" la palabra en fichas válidas; devuelve lista de tokens o vacía si falla.
- **private void construirDesdeArchivo(List<String> lineasArchivo)**: construye y minimiza el DAWG usando las palabras ordenadas.
- **public boolean buscarPalabra(String palabra)**: comprueba existencia exacta de la palabra en el DAWG.
- **public boolean buscarPrefijo(String prefijo)**: verifica si existe algún sufijo que comience con el prefijo dado.
- **public Nodo buscarUltimoNodo(String palabra)**: recorre el DAWG con la tokenización de la palabra y devuelve el último nodo alcanzado, o null si no existe.

3.2.1.11. Nodo

Nombre de la clase Nodo

Breve descripción de la clase
Nodo de un DAWG que almacena una ficha o letra, sus hijos y si marca el fin de una palabra válida.

Descripción de los atributos

- **letra** (String): token que identifica el nodo; null en la raíz.
- **hijos** (Map<String, Nodo>): mapa de nodos descendientes, paquete-access.
- **palabraValidaHastaAqui** (boolean): indica si aquí finaliza una palabra válida, paquete-access.

Descripción de los métodos

- **Nodo(String letra)**: crea un nodo con la ficha indicada.
- **Nodo()**: crea el nodo raíz (sin letra).
- **boolean esValida()**: true si marca fin de palabra.
- **String getLetra()**: devuelve la ficha o null.
- **Map<String, Nodo> getHijos()**: devuelve el mapa de hijos.
- **boolean equals(Object obj)**: igual si mismo token, estado y mismos hijos.
- **int hashCode()**: hash consistente con equals, combina estado, token y referencias de hijos.

3.2.1.12. Pair

Nombre de la clase Pair

Breve descripción de la clase
Contenedor genérico de dos valores (first, second).

Descripción de los atributos

- **first** (F): primer elemento del par.
- **second** (S): segundo elemento del par.

Descripción de los métodos

- **Pair(F first, S second)**: construye un par con los valores dados.
- **static <F,S> Pair<F,S> createPair(F first, S second)**: método fábrica para crear un par sin deducir tipos.
- **void setFirst(F first)**: asigna un nuevo valor a first.
- **void setSecond(S second)**: asigna un nuevo valor a second.
- **F getFirst()**: devuelve first.
- **S getSecond()**: devuelve second.
- **String toString()**: representación "(first, second)".
- **boolean equals(Object obj)**: compara ambos elementos para igualdad.
- **int hashCode()**: genera hash basado en first y second.

3.2.2. Controladores

3.2.2.1. CtrlJugador

Nombre de la clase CtrlJugador

Breve descripción de la clase

Gestiona la sesión del jugador activo, permitiendo iniciar/cerrar sesión, cambiar contraseña, eliminar perfil y actualizar puntuación, lanzando excepciones ante errores de sesión o contraseña.

Descripción de los atributos

- **jugadorActual** (Jugador): jugador que ha iniciado sesión, o null si no hay sesión.

Descripción de los métodos

- **boolean haySesion()**: devuelve true si jugadorActual != null.
- **void setJugadorActual(Jugador j)**: establece j como jugador activo.
- **void clearSesion()**: cierra la sesión (pone jugadorActual = null).
- **Jugador getJugadorActual()**: retorna jugadorActual, o lanza UsuarioNoEncontradoException si es null.
- **void cambiarPassword(String antigua, String nueva)**: cambia la contraseña tras validar la antigua; lanza UsuarioNoEncontradoException o PasswordInvalidaException.
- **void eliminarJugador(String password)**: elimina el perfil (cierra sesión) tras validar password; lanza UsuarioNoEncontradoException o PasswordInvalidaException.
- **void actualizarPuntuacion(int nuevosPuntos)**: actualiza jugadorActual.puntos si nuevosPuntos es mayor; lanza UsuarioNoEncontradoException si no hay sesión.

3.2.2.2. CtrlPartida

Nombre de la clase CtrlPartida

Breve descripción de la clase
Controlador de la lógica de partida: crea y carga partidas, gestiona turnos (incluyendo IA con Algoritmo), invoca validación de jugadas y actualiza estado (tablero, bolsa, puntuaciones).

Descripción de los atributos

- **partidaActual** (Partida): partida en curso.
- **dawg** (Dawg): diccionario DAWG para validar palabras.
- **validador** (Validador): validador de jugadas según reglas.
- **finTurno** (boolean): indicador de fin de turno.
- **tablero** (Tablero): referencia al tablero (no usado directamente).
- **bolsa** (Bolsa): referencia a la bolsa (no usado directamente).
- **isAlgoritmo** (boolean): true si juega IA.
- **algoritmo** (Algoritmo): instancia de IA para sugerir jugadas.
- **jugadorAlgoritmo** (boolean): indica si el segundo jugador es IA.

Descripción de los métodos

- **CtrlPartida()**: constructor que inicializa el validador.
- **void crearPartida(int modo, List<String> players, List<String> lineasArchivo, List<String> lineasArchivoBolsa, long seed, boolean jugadorAlgoritmo)**: crea DAWG y nueva Partida; configura IA según modo.
- **List<String> obtenerFichas()**: devuelve las letras en mano del jugador actual.

- **void cargarPartida(Partida partida)**: carga un objeto Partida existente.

- **Partida guardarPartida()**: retorna la Partida actual para persistencia.

- **int jugarScrabble(int opcion, String input)**: interpreta acciones de juego (colocar, quitar, pasar, fin de turno, cambiar fichas, IA) y lanza excepciones si hay error.
- **int getPuntosJugador1() / int getPuntosJugador2()**: devuelven puntuación de cada jugador.

- **int finTurno(boolean pasar, boolean algoritmo)**: finaliza o pasa turno, valida palabra, bloquea celdas, repone fichas y, si toca, ejecuta IA; retorna código de fin.

- **int finPartida(boolean abandono)**: resuelve la partida (por abandono o fin normal), ajusta puntos pendientes y devuelve el ganador (1 o 2).

- **Tablero obtenerTablero()**: obtiene el tablero de la partida actual.

- **private int jugarAlgoritmo()**: invoca el Algoritmo para calcular y colocar la mejor jugada IA y devuelve sus puntos.

3.2.2.3. CtrlDominio

Nombre de la clase CtrlDominio

Breve descripción de la clase
Coordina la capa de persistencia, la sesión de usuario y la lógica de partida, exponiendo operaciones de registro, autenticación, ranking, gestión de partidas y acceso a datos.

Descripción de los atributos

- **ctrlPersistencia** (CtrlPersistencia): controlador de acceso a almacenamiento de usuarios, partidas, diccionarios y bolsas.
- **ctrlJugador** (CtrlJugador): gestión de la sesión del jugador activo.
- **ctrlPartida** (CtrlPartida): gestión de la lógica de juego y partidas en curso.

Descripción de los métodos

- **CtrlDominio()**: constructor que inicializa los tres controladores.
- **void registrarJugador(String nombre, String password)**: crea y persiste un nuevo jugador, lanza UsuarioYaRegistradoException.
- **void crearUsuario(String nombre, String password)**: alias de registrarJugador.
- **Jugador obtenerJugador(String nombre)**: recupera un jugador por nombre o lanza UsuarioNoEncontradoException.
- **void iniciarSesion(String nombre, String password)**: autentica y abre sesión; lanza UsuarioNoEncontradoException o AutenticacionException.
- **void cerrarSesion()**: cierra la sesión actual.
- **boolean haySesion()**: indica si hay sesión activa.
- **String getUsuarioActual()**: devuelve el nombre del jugador en sesión o null.
- **int getPuntosActual()**: devuelve la puntuación del jugador en sesión o 0.
- **void cambiarPassword(String antigua, String nueva)**: cambia la contraseña y actualiza persistencia; lanza PasswordInvalidaException.

- **void eliminarUsuario(String password)**: elimina perfil y persistencia tras validar contraseña; lanza PasswordInvalidaException.

- **List<Map.Entry<String,Integer>> obtenerRanking()**: retorna lista de pares (nombre, puntos) ordenada.

- **int getPosition(String nombre)**: devuelve posición en ranking o lanza UsuarioNoEncontradoException.

- **int getPosicionActual()**: posición en ranking del usuario en sesión, o –1.

- **void iniciarPartida(int modo, String n1, String n2, String idDiccionario, long seed, boolean jugadorAlgoritmo)**: configura y arranca una nueva partida.

- **int jugarScrabble(int modo, String jugada)**: delega la jugada, actualiza puntuación en sesión y persistencia; lanza excepciones de tablero o ficha.

- **Tablero obtenerTablero() / List<String> obtenerFichas() / int getPuntosJugador1() / int getPuntosJugador2()**: exponen datos de la partida actual.

- **void guardarPartida(String id) / void cargarPartida(String id) / void cargarUltimaPartida() / void eliminarPartidaGuardada(String id)**: gestionan persistencia de partidas.

- **int getTableroDimension() / String getLetraCelda(int fila, int col) / String getBonusCelda(int fila, int col)**: helpers para el driver de interfaz.

- **Set<String> obtenerIDsDiccionarios() / Set<String> obtenerIDsBolsas() / void crearDiccionario(String id, List<String> palabras) / void crearBolsa(String id, Map<String,int[]> datos) / void eliminarIdiomaCompleto(String id) / List<String> getDiccionario(String id) / List<String> getBolsa(String id)**: operaciones de gestión de recursos de diccionarios y bolsas.

3.2.3. Drivers

3.2.3.1. Gestión de Diccionarios y Bolsas

Nombre de la clase DriverGestionDiccionariosBolsas

Breve descripción de la clase
Driver de consola que, mediante un menú interactivo, permite consultar, probar, crear y eliminar recursos de diccionarios y bolsas usando la capa de dominio (CtrlDominio).

Descripción de los atributos

- **sc** (Scanner): escáner estático para leer la entrada del usuario.
- **ctrl** (CtrlDominio): instancia de controlador de dominio para gestionar diccionarios y bolsas.

Descripción de los métodos

- **menuPrincipal()**: muestra el menú principal y redirige a la opción seleccionada.
- **consultarRecursos()**: muestra en tabla los IDs de diccionarios y bolsas registrados.
- **juegoPruebas()**: ejecuta automáticamente pruebas de creación y eliminación de recursos con IDs predefinidos.
- **gestionRecursos()**: menú para eliminar o añadir recursos manualmente.
- **eliminarRecurso()**: pide un ID y elimina diccionario y bolsa asociados; maneja IOException.
- **anadirRecurso()**: submenú para añadir recursos desde directorio o desde teclado.
- **anadirDesdeDirectorio()**: solicita ruta de directorio, valida archivos y crea recursos leyendo ficheros.
- **anadirDesdeTeclado()**: solicita ID, palabras y líneas de bolsa al usuario y crea los recursos.
- **leerArchivoTexto(String ruta)**: lee un fichero de texto y devuelve lista de líneas no vacías.
- **leerArchivoBolsa(String ruta)**: lee un fichero de bolsa formateado (“Ficha Cantidad Puntuacion”) y devuelve un mapa de datos.

3.2.3.2. Gestión de Usuarios

Nombre de la clase DriverGestionUsuarios

Breve descripción de la clase
Driver de consola que gestiona la sesión de usuarios: permite iniciar/cerrar sesión, registrarse, ver ranking, cambiar contraseña y eliminar perfil, además de ejecutar pruebas automatizadas.

Descripción de los atributos

- **ctrl** (CtrlDominio): controlador de dominio para operaciones sobre usuarios.
- **sc** (Scanner): lector estático de entrada estándar.

Descripción de los métodos

- **menuPrincipal()**: muestra opciones para iniciar sesión, registrarse, ejecutar pruebas o salir.
- **menuUsuario()**: muestra opciones para ver cuenta, ranking, cerrar sesión o salir.
- **iniciarSesion()**: lee usuario/contraseña, llama a CtrlDominio para autenticar y muestra datos de bienvenida.
- **registrarse(boolean inicio)**: solicita nuevo usuario y contraseña, crea cuenta y opcionalmente inicia sesión.
- **verRanking()**: obtiene y muestra el ranking global desde CtrlDominio.
- **subMenuCuenta()**: presenta datos de la cuenta activa y opciones para cambiar contraseña o eliminar perfil.
- **cambiarPassword()**: solicita contraseñas, valida coincidencia y llama a CtrlDominio para cambiarla.
- **eliminarPerfil()**: confirma intención, solicita contraseña y elimina perfil vía CtrlDominio.
- **juegoPruebas()**: ejecuta creación, login, cambio de contraseña y eliminación para un conjunto de usuarios de prueba.

3.2.3.3. Gestión de Partidas

Nombre de la clase DriverGestionPartidas

Breve descripción de la clase
Driver de consola para gestionar partidas de Scrabble: crear, cargar, listar, eliminar partidas guardadas, arrancar nuevas partidas y ejecutar pruebas automatizadas.

Descripción de los atributos

- **cd** (CtrlDominio): controlador de dominio para operaciones de partida.
- **MODO1**, **MODO2** (String): constantes de texto para modo de juego (1 Jugador / 2 Jugadores).
- **modo**, **idDiccionario**, **nombre1**, **nombre2** (String): variables estáticas de configuración de nueva partida.

Descripción de los métodos

- **menuPrincipal()**: menú inicial con opciones para crear, seguir, cargar, gestionar y probar partidas.
- **subMenuGestionGuardadas()**: lista partidas guardadas y permite cargar o eliminar.
- **subMenuCrearPartida()**: menú guiado para seleccionar modo, diccionario y nombres, y lanzar la partida.
- **datosCompletos()**: comprueba que se han seleccionado modo, diccionario y nombres necesarios.
- **iniciarYJugar()**: genera semilla aleatoria, llama a CtrlDominio para iniciar partida y pasa al submenú de juego.
- **elegirModo()**: permite al usuario escoger entre 1 o 2 jugadores.
- **elegirDiccionario()**: muestra diccionarios disponibles y guarda la selección.
- **introducirNombres()**: lee desde teclado nombre(s) de jugador(es).
- **subMenuCargarPartida()**: muestra lista de partidas guardadas y carga la seleccionada.
- **subMenuPartida()**: tras iniciar/cargar, muestra tablero, fichas y opciones para abandonar, guardar o continuar.
- **imprimirTablero()**: pinta en consola el estado del tablero con letras y bonos.
- **juegoPruebas()**: recorre modos y diccionarios, prueba creación, guardado, carga y eliminación de partidas.

3.2.3.4. Gestión de Juego

Nombre de la clase DriverGestionJuego

Breve descripción de la clase
Driver de consola para probar partidas contra la IA y manejar menús de juego: registro, creación de partida, interacción por turnos, comandos de colocación/remoción/cambio de fichas, pasar/acabar turno y pruebas automatizadas.

Descripción de los atributos

- **cd** (CtrlDominio): controlador de dominio para acceder a la lógica de usuario y juego.
- **nombreJugador** (String): nombre de usuario fijo para pruebas ("A").
- **contrasena** (String): contraseña fija para pruebas ("A").
- **fin** (int): código de fin de partida (0 = continuar, 1 o 2 = ganador).
- **modo** (String): modo de juego seleccionado ("1 Jugador"/"2 Jugadores").
- **idioma** (String): identificador de diccionario/bosla usado.

Descripción de los métodos

- **void mostrarFichas()**: imprime las fichas actuales del jugador.
- **void mostrarTablero()**: dibuja el tablero con letras y bonificaciones.
- **void mostrarEstado()**: muestra puntos de ambos jugadores, tablero y fichas.
- **void menuPartida()**: menú de turno con opciones de añadir/quitar/cambiar fichas, pasar y acabar turno, IA y salir.
- **void finalPartida(int fin)**: muestra el ganador según el código de fin.
- **void menuPrincipal()**: menú inicial con "Jugar", "Juego de pruebas" y "Salir".
- **void subMenuCrearPartida()**: opciones para seleccionar modo e idioma y lanzar la partida.
- **void elegirModo()**: define el modo (1 o 2 jugadores) y vuelve a crearPartida.

- **void elegirIdioma()**: lista diccionarios disponibles y guarda la selección.

- **void subMenuAnadir()**: pide entrada para añadir ficha y llama a jugarScrabble(1).

- **void subMenuQuitar()**: pide coordenadas para quitar ficha y llama a jugarScrabble(2).

- **void subMenuCambiar()**: pide fichas a cambiar y llama a jugarScrabble(5).

- **void subMenuSalir()**: opciones para abandonar, guardar o volver atrás.

- **void menuJuegoPruebas()**: elige entre tres modos de pruebas (solo/duo/IA vs IA).

- **void juegoPruebas1()**, **void juegoPruebas2()**, **void juegoPruebas3()**: scripts de prueba que simulan partidas ejecutando secuencia de jugadas y guardados.

3.2.3.5. Driver Scrabble

Nombre de la clase
DriverScrabble

Breve descripcion
Driver integral de la capa de presentacion que ofrece menus interactivos para gestionar usuarios, diccionarios y bolsas, partidas y el flujo completo del juego de Scrabble.

Atributos

- sc (Scanner): lector de entradas por consola.
- ctrl (CtrlDominio): punto de acceso a la logica del dominio.

Metodos

- **menuPrincipal():** muestra opciones de iniciar sesion, registrarse o salir.
- **menuUsuario():** menu tras login con opciones de gestion de cuenta, diccionarios/bolsas, partidas, cerrar sesion o salir.
- **subMenuCuenta():** opciones sobre la cuenta actual (cambiar password, ver ranking, eliminar perfil, volver).
- **cambiarPassword():** pide contrasena actual y nueva, valida igualdad y aplica cambio.
- **verRanking():** muestra ranking global de puntos (lanza RankingVacioException si no hay datos).
- **eliminarPerfil():** confirma y elimina perfil tras validacion.
- **iniciarSesion():** solicita usuario y password, inicia sesion e imprime bienvenida.
- **registrarse()**: solicita nuevo usuario y password, registra y abre sesion.
- **subMenuDiccionarios()**: menu para consultar, anadir o eliminar diccionarios y bolsas.
- **consultarRecursos()**: lista IDs de diccionarios y bolsas con sus ficheros.
- **addResourceMenu()**: submenu para anadir recursos desde directorio o teclado.

- **addFromDirectory():** importa un diccionario y bolsa desde una carpeta existente.

- **addFromKeyboard():** crea diccionario y bolsa leyendo lineas de consola.

- **deleteResource()**: elimina diccionario y bolsa por ID.

- **subMenuPartidas()**: menu para crear, cargar ultima, cargar guardada o ver ranking.

- **crearPartida()**: seleccion de modo, diccionario y jugadores, arranque de partida.

- **cargarUltimaPartida():** recupera la partida mas reciente guardada.

- **cargarPartidaGuardada()**: lista y carga partidas previamente guardadas.

- **menuJuego():** muestra estado de la partida, tablero y permite anadir, quitar, cambiar fichas, pasar, finalizar turno, guardar o abandonar.

- **mostrarTablero()**: imprime en consola la matriz de celdas con letras y bonificadores.

- **subMenuAnadir**(), **subMenuQuitar()**, s**ubMenuCambiar()**: handlers que leen parametros de ficha/posicion y llaman a ctrl.jugarScrabble(...).

3.2.4. Excepciones

3.2.4.1. UsuarioNoEncontradoException

Indica que se ha solicitado un usuario que no existe en el sistema. Se lanza, por ejemplo, al iniciar sesión con un nombre no registrado o al intentar operar sobre una cuenta inexistente.

3.2.4.2. UsuarioYaRegistradoException

Señala que se está intentando crear un perfil con un nombre que ya existe. Aparece durante el registro de un nuevo usuario y detiene el proceso para que el sistema informe de que debe elegirse otro identificador.

3.2.4.3. PasswordInvalidaException

Informa de que la clave proporcionada para cambiar contraseña o para confirmar la eliminación de la cuenta no coincide con la almacenada.

3.2.4.4. BolsaNoEncontradaException

Indica que se ha intentado acceder a una bolsa de fichas que no está registrada en el sistema.

3.2.4.5. BolsaYaExistenteException

Aparece cuando se quiere registrar una nueva bolsa de fichas usando un identificador que ya corresponde a otra existente.

3.2.4.6. DiccionarioNoEncontradoExcpection

Señala que se ha solicitado un diccionario que no existe en el sistema, ya sea al iniciar una partida o al intentar visualizarlo.

3.2.4.7. DiccionarioYaExistenteException

Se lanza cuando se intenta crear un nuevo diccionario con un identificador que ya está en uso en el sistema.

3.2.4.8. PartidaYaExistenteException

Se lanza al intentar guardar una partida con un identificador que ya existe, para evitar sobrescribir datos sin confirmación.

3.2.4.9. PartidaNoEncontradaException

Se lanza al intentar cargar o eliminar una partida con un identificador que no está presente en las partidas guardadas.

3.2.4.10. NoHayPartidaGuardadaException

Indica que se ha solicitado cargar la última partida guardada, pero no existe ninguna partida previa en el sistema.

3.2.4.11. PuntuacionInvalidaExcecpion

Se lanza cuando se intenta asignar una puntuación negativa a un jugador, lo cual no está permitido por las reglas del sistema de ranking.

3.2.4.12. RankingVacioExcepcion

Indica que no hay jugadores registrados con puntuación positiva, por lo que el ranking global está vacío y no puede mostrarse.

3.2.4.13. ComandoInvalidoException

Señala que el formato o el contenido del comando introducido por el jugador es incorrecto, como por ejemplo al intentar poner fichas o coordenadas inválidas.

3.2.4.14. PalabraInvalidaException

Se lanza cuando la palabra formada durante una jugada no cumple las reglas del juego (por ejemplo, no está en el diccionario, no conecta con el tablero, etc.). El mensaje mostrado al usuario es siempre “Palabra Incorrecta”.

4. Relación de las clases implementadas por los miembros del grupo

Clase	Implementado por
CtrlDominio	Alex + Soukaina
CtrlJugador	Alex + Soukaina
CtrlPartida	Alex
Jugador	Alex
Partida	Alex
Tablero	Soukaina
Celda	Soukaina
Ficha	Soukaina
DAWG	Soukaina
Nodo	Soukaina
Validador	Alex
Algoritmo	Alex
CtrlPersistencia	Soukaina + Alex
DriverGestionUsuarios	Soukaina
DriverGestionPartidas	Soukaina
DriverGestionDiccionarioBolsas	Soukaina
DriverGestionJuego	Alex
DriverScrabble	Soukaina + Alex
Excepciones	Soukaina

5. Estructura de datos y algoritmos utilizados

5.1. Estructuras de datos

5.1.1. Pair<L,R>

Contenedor genérico que solo agrupa dos valores (**Left** y **Right**) sin más semántica para cuando se necesita devolver o pasar un par de objetos sin crear una clase más.

5.1.2. List<Object>

La hemos usado en la clase Bolsa, el campo **List<Ficha> conjuntoDeFichas** mantiene la reserva activa de letras: una colección ordenada y mutable que admite duplicados

5.1.3. DAWG (Grafo acíclico dirigido de palabras)

La clase **DAWG** representa un grafo acíclico dirigido construido a partir de un conjunto de palabras válidas. Internamente, utiliza un nodo raíz, una colección de fichas válidas (HashSet), un registro de subgrafos (HashMap) y nodos conectados recursivamente (Map<String, Nodo>). Esta estructura permite validar y buscar palabras de forma eficiente, aprovechando la compresión estructural del grafo al compartir sufijos comunes entre palabras.

5.1.4. Map<String, Nodo> (hijos de un Nodo)

Dentro de la clase **Nodo**, que forma parte del DAWG (Directed Acyclic Word Graph), se utiliza un **Map<String, Nodo>** para almacenar los hijos del nodo actual. Esta estructura permite representar conexiones desde un nodo hacia sus siguientes posibles letras. La elección de un mapa proporciona una forma eficiente de navegación y construcción del grafo al permitir acceso rápido por clave (letra).

5.1.5. HashSet<String> (fichasValidas)

En la clase del Dawg, se ha utilizado un **HashSet<String>** para almacenar las fichas válidas según los datos de configuración de la bolsa. Esta estructura permite validar de forma eficiente (O(1) promedio) si una ficha introducida por el usuario o generada por el algoritmo es válida. Es especialmente útil durante la construcción del DAWG y en el proceso de búsqueda y validación de palabras.

5.1.6. HashMap<String, Nodo> (Registro de los subgrafos)

La clase Dawg emplea un **HashMap<String, Nodo>** llamado registro, que actúa como una tabla de memorización durante la construcción del grafo. Cada entrada en este mapa permite detectar subgrafos ya existentes que pueden ser reutilizados, lo que evita duplicación de nodos y asegura que el grafo resultante sea compacto, eficiente y sin redundancias.

5.1.7. Nodo (Estructura recursiva para el DAWG)

La clase **Nodo** es una estructura de datos recursiva diseñada para representar los vértices de un DAWG. Cada nodo puede tener múltiples hijos (otras instancias de **Nodo**) y un indicador booleano que señala si ese nodo corresponde al final de una palabra válida.

5.1.8. Map<String, Jugador> (Usuarios registrados)

Se ha utilizado un **Map<String, Jugador>** para almacenar los usuarios registrados del sistema, donde la clave corresponde al nombre del jugador y el valor es el objeto Jugador asociado. Esta estructura permite un acceso rápido a los datos de cada jugador, facilita la comprobación de existencia y agiliza operaciones como el registro, actualización o eliminación de usuarios. Su uso está fuertemente ligado al sistema de persistencia de usuarios.

5.1.9. NavigableSet<Jugador> (Mediante TreeSet<Jugador> con Comparator personalizado)

Para mantener ordenado el ranking de jugadores se ha empleado un **TreeSet<Jugador>** con un **Comparator** personalizado. Este comparador ordena a los jugadores en primer lugar por puntuación (de mayor a menor) y, en caso de empate, alfabéticamente por nombre. Esta estructura garantiza la unicidad de los elementos, y al estar automáticamente ordenada, resulta ideal para representar un ranking competitivo sin necesidad de reordenamientos manuales.

5.1.10. Map<String, Partida> (Partidas en memoria)

La colección de partidas en memoria se ha gestionado mediante un **Map<String, Partida>**, donde cada clave representa el identificador único de una partida y el valor es el objeto Partida correspondiente. Esta estructura permite guardar, cargar y eliminar partidas de forma flexible y eficiente, lo cual es fundamental para permitir partidas múltiples, recuperación del estado de juego y gestión de partidas guardadas.

5.1.11. Map<String, List<String>> (Bolsas y Diccionarios)

Los diccionarios y las bolsas se han representado mediante dos estructuras **Map<String, List<String>>**. Cada clave corresponde al identificador del idioma, tema o simplemente un ID, mientras que el valor es una lista de cadenas de texto que representa el contenido del recurso: palabras válidas para el diccionario o configuraciones de letras para la bolsa. Esta representación es simple, eficiente y directamente compatible con operaciones de entrada/salida desde archivos.

5.2. Algoritmos

5.2.1. Grafo Dirigido Acíclico de Palabras

1. Definición del DAWG

Un **DAWG** (o en castellano Grafo Dirigido Acíclico de Palabras) es una estructura de datos para almacenar eficientemente un gran diccionario de palabras, compartiendo sufijos comunes, el cual usaremos como diccionario en nuestro Scrabble. S

Su principal característica es que minimiza la redundancia al fusionar subárboles idénticos, reduciendo drásticamente el número de nodos y, por tanto espacio en la memoria y garantizando una rapidez de búsqueda.

2. Componentes básicos

- **Nodo**: representa un estado en el grafo; contiene un mapa de hijos (**hijos**) indexado por fichas (de tipo String) y un booleano **palabraValidaHastaAqui**.

- **Fichas válidas**: conjunto de cadenas (por ejemplo, “A”, “CH”, “LL”) que el DAWG acepta como unidades de tokenización.

- **Registro de minimización**: un **HashMap<Nodo, Nodo>** que mantiene nodos ya procesados y permite fusionar rápidamente nuevos nodos equivalentes.

3. Flujo principal del DAWG

1. **Carga de fichas (cargarFichasValidas)**

    - Lee un listado y extrae la primera columna como ficha y guarda en un **HashSet**, el fichero tiene que seguir las indicaciones mencionadas por el profesorado.

    - Calcula la longitud máxima **L** que pueden tener las fichas.

2. **Tokenización de una palabra**

    - Recorre la palabra (longitud n) de izquierda a derecha.

    - Para cada posición, prueba subcadenas decrecientes de tamaño hasta L (lo máximo que puede tener la ficha, parámetro calculado previamente en cargarFichasVálidas), buscando la más larga que exista en el conjunto de fichas.

    - Si no encuentra ninguna, la tokenización falla; en caso contrario, avanza el índice y repite.

3. **Construcción incremental minimizada**
    - Asume que la lista de palabras (W palabras, una por cada línea) ya viene ordenada lexicográficamente.
    - Para cada nueva palabra:
        1. **Tokenizar** (coste ~O(m·L), m es la longitud de la palabra).
        2. **Detectar prefijo común** con la palabra anterior (O(min(m, m□))). por lo que como máximo se hacen comparaciones igual al tamaño de la palabra más corta
        3. **Minimizar** todos los nodos correspondientes al sufijo que ya no comparten prefijo: por cada nodo sobrante, comprueba en el registro; si existe un equivalente, "apunta" al nodo compartido, si no, lo añade al registro. Esto toma tiempo proporcional al número de nodos a procesar, O(s).
        4. **Insertar** los tokens restantes como nuevos nodos en el grafo (O(t), con t = tokens nuevos).
    - Tras procesar todas las palabras, minimiza el camino de la última palabra de la misma forma.

Ejemplo simple para explicar el proceso + coste, "CASA" → "CASO"

Prefijo común

- Anterior = **C A S A**
- Nueva = **C A S O**
- Prefijo = **C A S** (3 letras/tokens/nodos comunes)

Minimizar sufijo anterior

- Sufijo de "CASA" más allá del prefijo: **A**
- Compruebo en el registro si ya existe un nodo equivalente a ese "A" terminal.
- Si existe, lo apunto; si no, lo guardo.
- Coste**:** O(s)=O(1) (s = 1 nodo)

Insertar sufijo nuevo

- Sufijo de "CASO" más allá del prefijo: **O**
- Creo un nuevo nodo "O" marcado como final y lo enlazo.
- Coste: O(t)=O(1) (t = 1 nodo)

```
Unset
(raíz) --C--> [C] --A--> [A] --S--> [S] --A* (CASA)
                                  \
                                   --> O* (CASO)
```

4. **Búsquedas**
    - **Palabra completa**: tokeniza (O(m·L)) y recorre los hijos; al final comprueba palabraValidaHastaAqui. (m es la longitud de la palabra).
    - **Prefijo**: igual que palabra, pero sólo importa llegar a un nodo válido.
    - **Último nodo**: devuelve el nodo alcanzado tras tokenizar; null si algún paso falla.

4. Análisis de complejidad

- **Parámetros clave**
    - W: número de palabras a cargar.
    - m: longitud media de cada palabra (en caracteres).
    - L: longitud máxima de ficha (token).

- **cargarFichasValidas**
    - Recorre todas las líneas (digamos M): **O(M)**.

- **tokenizarPalabra**
    - En cada uno de sus m caracteres prueba hasta L subcadenas y busca en un HashSet (O(1) medio): **O(m·L)**.

- **construirDesdeArchivo**
    - Para cada palabra (total W):

        1. Tokenizar: **O(m·L)**
        2. Prefijo común: **O(m)**
        3. Minimización de sufijos: **O(m)**
        4. Inserción de nuevos nodos : **O(m)**

    - Como m y L suelen acotarse (longitud razonable de palabra y ficha), el coste real crece **aproximadamente lineal** en el número de palabras: **O(W·m·L)** (o, para m,L constantes, entonces **O(W)**).

- **buscarPalabra / buscarPrefijo / buscarUltimoNodo**
    - Cada uno tokeniza (O(m·L)) y desciende por a lo sumo m hijos (O(1) medio por acceso): **O(m·L)**.

5.2.2. Algoritmo de resolución del Scrabble (Jugador máquina)

1. Explicación

El algoritmo tiene como objetivo identificar la mejor palabra disponible dado un tablero con unas fichas colocadas y una lista de máximo 7 fichas del jugador, que proporcione la mayor cantidad de puntos de una manera eficiente. Para ello, combina el valor intrínseco de cada ficha con las bonificaciones del tablero (letras dobles/triples, palabras dobles/triples y el bono de 50 puntos por usar todas las fichas).

Está inspirado en la estrategia de Appel & Jacobson (1988), que demuestra cómo, gracias a la simetría del tablero, basta plantear la búsqueda para una sola dirección (por ejemplo, horizontal) y, con un simple cambio de orientación, repetirla en vertical. De este modo, el problema se reduce a una sola dimensión: generamos movimientos sobre filas (o columnas, trasponiendo el tablero) y aprovechamos esa abstracción a lo largo de todo el proceso.

La estrategia del algoritmo consiste en partir de las llamadas **celdas ancla** ("anchors"), que son las casillas vacías adyacentes a fichas ya existentes en el tablero. A partir de cada ancla se exploran primero los posibles prefijos situados antes de ella usando únicamente fichas de la mano y después se extienden esos prefijos hacia la derecha, letra a letra, gracias a la estructura de datos DAWG que actúa como diccionario en memoria. En cada paso de la extensión se valida la formación de palabras perpendiculares (los "cross-checks"), y siempre que se alcanza un nodo terminal del DAWG en una casilla vacía se calcula la puntuación completa de la palabra resultante, actualizando la mejor jugada si supera el récord previo.

2. Componentes básico

- **DAWG.** Un grafo acíclico y mínimo que alberga todo el diccionario en memoria. Cada nodo representa un prefijo válido y cada arista, una letra o diptongo; los nodos terminales señalan palabras completas.
- **Tablero.** El tablero es una matriz de 15 × 15 celdas que pueden contener fichas y poseen bonificadores individuales de letra o palabra. El algoritmo interactúa con él para consultar si cada casilla está vacía u ocupada, leer las fichas ya colocadas adyacentes, aplicar los multiplicadores correspondientes y calcular también los puntos de las palabras perpendiculares que se formen al añadir nuevas letras.
- **Lista de fichas del jugador.** La mano del jugador consiste en un máximo de siete fichas, cada una identificada por su letra (o blank) y un valor en puntos. Durante la exploración, las fichas se retiran y reponen de esta lista para probar todas las combinaciones posibles que encajen en el DAWG.

3. Flujo principal del Algoritmo.

El núcleo de la búsqueda se organiza en cinco fases claramente diferenciadas. A continuación se describen en detalle cada una de ellas, numeradas como 3.1 a 3.5,

aunque su ejecución se encadena de forma continua hasta hallar la jugada de mayor puntuación.

3.1. **Búsqueda de anclas.** El algoritmo comienza con un recorrido completo del tablero para identificar las celdas ancla, es decir, aquellas casillas vacías que poseen al menos un vecino ocupado (ya sea a la izquierda, derecha, arriba o abajo). Todas las posiciones que cumplen este criterio se almacenan en una lista de anclas; este conjunto de puntos de partida asegura que cada palabra generada se conectará con fichas ya presentes en el tablero, evitando la exploración de jugadas aisladas que no aportan valor. En caso de no encontrar ninguna, significa que el algoritmo está en el turno inicial, cuando el tablero está vacío, en ese caso, usará como ancla el centro del tablero.

3.2. **Construcción inicial de prefijos.** Para cada ancla de la lista, se determina el prefijo fijo situado inmediatamente antes de ella en la dirección de juego (horizontal o, más adelante, vertical). Si existen fichas adyacentes en esa dirección, se extraen de izquierda a derecha hasta llegar a una casilla vacía o al límite del tablero, formando un fragmento que queda fuera del control de la mano. En ausencia de letras previas, el algoritmo calcula cuántas casillas vacías consecutivas hay antes del ancla para establecer un "límite" que indicará cuántas fichas de la mano se podrán usar al crear prefijos.

3.3. **Generación de prefijos en la izquierda/arriba (Before Part).** Con el espacio libre delimitado, arranca la función **beforePart**, que explora recursivamente todas las combinaciones posibles de fichas de la mano para formar prefijos a la izquierda del ancla. Partiendo del nodo raíz del DAWG y con un contador que representa las casillas disponibles, el algoritmo prueba una a una las letras (o blanks) que aún quedan, avanza por la arista correspondiente en el grafo y reduce el límite en uno, retirando temporalmente la ficha. Cada vez que se coloca una nueva letra, se repite el proceso hasta agotar los espacios o las fichas, y en cada paso se invoca la fase de extensión (3.4) para prolongar ese prefijo hacia la derecha.

3.4. **Extensión hacia la derecha/abajo y validación (Extend After).** La función extendAfter recibe un prefijo ya formado, su nodo asociado en el DAWG y la posición de la casilla vacía situada tras la última letra. A partir de ahí, recorre cada arista hija del nodo actual, comprueba si la letra correspondiente está disponible en la mano y, antes de colocarla,

ejecuta un chequeo perpendicular ("cross-check") para asegurar que la palabra vertical que se formaría es válida. Si el cruce resulta aceptable, avanza al siguiente nodo del DAWG y casilla del tablero, acumula provisionalmente la puntuación de esa letra (incluyendo bonificadores de letra y palabra) y retira la ficha de la mano. Cuando alcanza un nodo terminal en una casilla vacía, significa que ha construido una palabra completa y válida, momento en el cual se pasa al cálculo definitivo de puntos.

3.5. **Cálculo de puntuación y selección final**. Al completarse una palabra válida, el algoritmo recorre toda la secuencia de letras recién colocadas junto a las que ya había en el tablero para aplicar puntualmente los multiplicadores de casilla, sumar el valor base de cada ficha y añadir el bono de 50 puntos si se han usado todas las fichas de la mano. Este total se compara con la mejor puntuación registrada hasta el momento; si la supera, se actualiza la jugada óptima guardando el conjunto de letras y sus posiciones. Una vez procesadas todas las anclas en ambas orientaciones (horizontal y vertical), el algoritmo devuelve finalmente la colocación de mayor puntuación junto con su valor total.

4. Análisis de coste y complejidad.

Parámetros clave

- B: número de fichas del jugador (máximo 7).
- **L**: espacios libres contiguos a la izquierda/arriba de un ancla.
- **R:** espacios libres contiguos a la derecha/abajo de un ancla

- **Búsqueda de anclas.** El tablero es siempre de 15×15 casillas (225 posiciones). Para localizar las anclas, el algoritmo recorre cada celda y, en O(1), por cada celda, comprueba sus cuatro vecinos. Por tanto, este paso tiene coste O(1) "amortizado" en un tablero de tamaño fijo, equivalente a un escaneo lineal de 225 celdas. En la práctica, su tiempo es constante e insignificante comparado con las fases de backtracking.
- **Construcción inicial de prefijos.** Por cada ancla detectada, el algoritmo retrocede en la dirección de juego (hacia la izquierda si es horizontal, hacia arriba si es vertical) hasta toparse con una ficha o el borde del tablero. Este avance atrás consume, en el peor de los casos, hasta 14 pasos (una fila completa menos la propia ancla), por lo que su coste es O(L) con $L \leq 15$ constante. Incluso asumiendo el caso extremo de 225 anclas, el tiempo global de esta fase sigue siendo constante sobre un tablero de tamaño 15x15.
- **Generación de prefijos izquierdos (Before Part).**

  La función *beforePart* realiza un backtracking sobre el DAWG y la mano del jugador para producir todos los prefijos que encajen antes de cada ancla. Si B es el número de fichas en mano ($B \leq 7$) y L el número de casillas libres contiguas ($R \leq 7$), el coste teórico sin podas sería O(B^L). Sin embargo, la estructura mínima del DAWG elimina instantáneamente cualquier letra que no pueda continuar un

prefijo válido, y la reducción progresiva de fichas disminuye drásticamente el número de opciones en cada nivel. En la práctica, la exploración se limita a muy pocos nodos y resulta extremadamente rápida.

- **Extensión hacia la derecha y validación (Extend After).** *extendAfter* retoma cada prefijo y lo prolonga letra a letra hacia adelante, comprobando en cada paso dos condiciones: que la letra exista en la mano y que la palabra perpendicular formada sea válida (mediante una llamada dinámica a recorrerDireccion). Si R es el número de casillas libres a la derecha ($R \leq 15$), la complejidad teórica sería O(B^R), pero de nuevo las podas del DAWG y la eliminación inmediata de cruces inválidos (con la función recorrerDireccion) reducen el espacio de búsqueda a unas decenas o centenares de nodos por ancla. Este chequeo dinámico de cruces sacrifica algo de tiempo frente a los bit-vectors precomputados del paper original, pero aporta mayor flexibilidad a la hora de validar cualquier configuración de tablero.
- **Cálculo de puntuación y selección final**. Cada vez que el algoritmo completa una palabra válida, recorre todas sus letras para aplicar los multiplicadores de letra y de palabra y sumar el valor individual de cada ficha. Este proceso consume un tiempo proporcional a la longitud de la palabra evaluada. Como normalmente solo se consideran unas pocas centenas de jugadas en total, el coste agregado de calcular la puntuación es lineal y resulta prácticamente insignificante frente al esfuerzo del backtracking.